TREE

Non-Linear Data structure

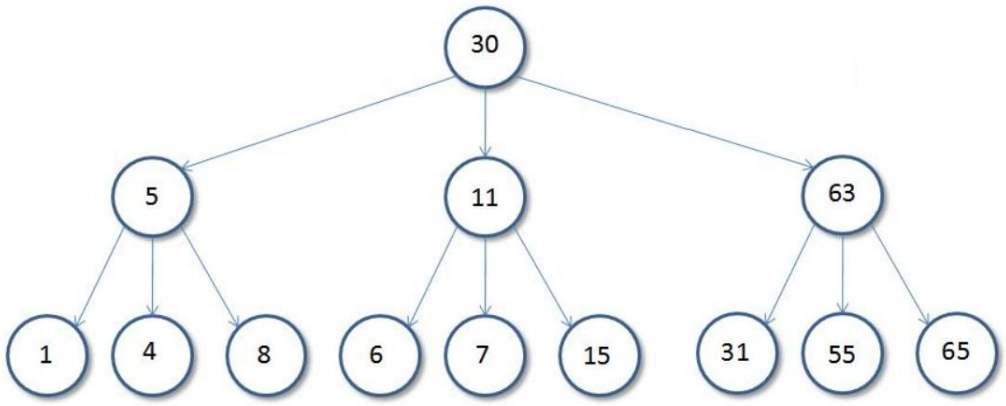
NOVEMBER 12, 2020

Table of Contents

Tree data structure

> *A Tree is a non-linear data structure where each node is connected to a number of nodes with the help of pointers or references.*

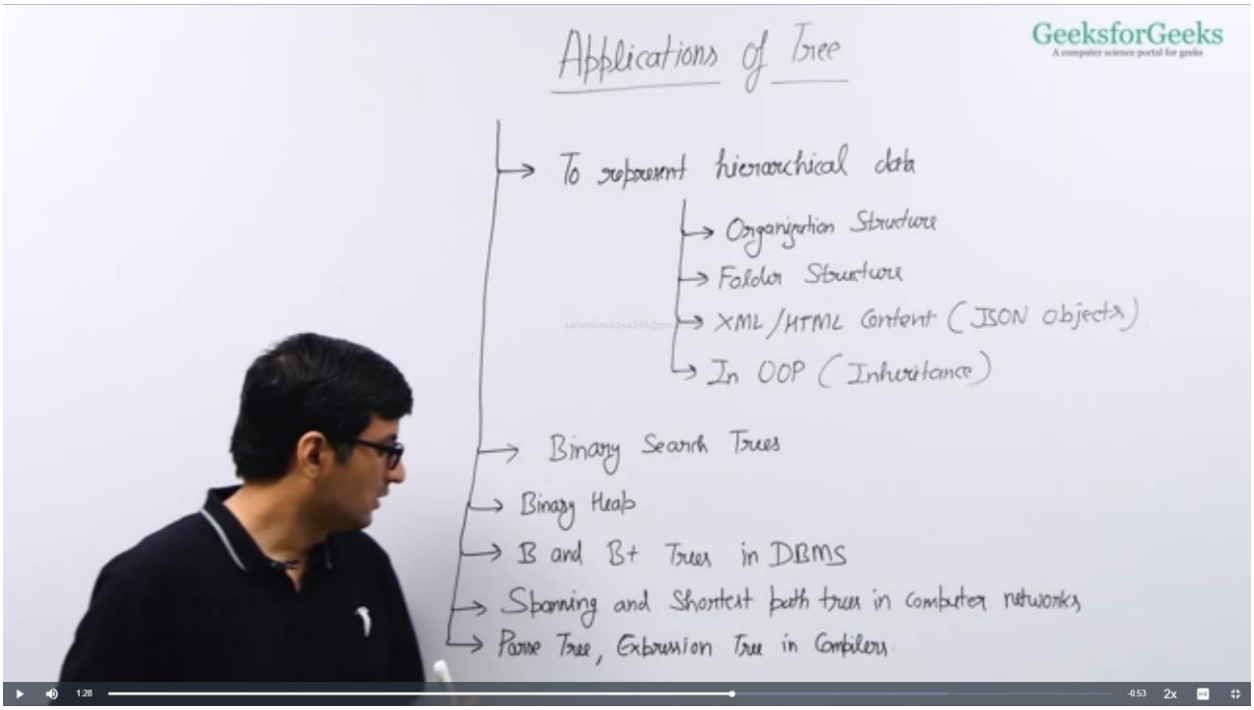
A Sample tree is as shown below:



Basic Tree Terminologies:

- **Root**: The root of a tree is the first node of the tree. In the above image, the root node is the node 30.
- **Edge**: An edge is a link connecting any two nodes in the tree. For example, in the above image there is an edge between node **11** and **6**.
- **Siblings**: The children nodes of same parents are called siblings of each other. That is, the nodes with same parents are called siblings. In the above tree, nodes *5, 11 and 63* are siblings.
- **Leaf Node**: A node is said to be the leaf node if it has no children. In the above tree, node **15** is one of the leaf node.
- **Height of a Tree**: Height of a tree is defined as the total number of levels in the tree or the length of the path from the root node to the node present at the last level. The above tree is of height **2**.

Application of Tree



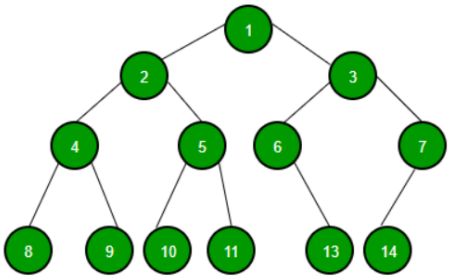
*Study all the applications in detail

Binary tree

Binary Tree

> *A Tree is said to be a Binary Tree if all of its nodes have atmost 2 children. That is, all of its node can have either no child, 1 child or 2 child nodes.*

Below is a sample **Binary Tree**:



Properties of Binary tree

Properties of a Binary Tree:

1. *The maximum number of nodes at level 'l' of a binary tree is (2^{l-1}).* Level of root is 1.

   This can be proved by induction.
   For root, l = 1, number of nodes = $2^{1-1} = 1$
   Assume that the maximum number of nodes on level l is 2^{l-1}.
   Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$.
2. *Maximum number of nodes in a binary tree of height 'h' is ($2^h - 1$).*
   Here height of a tree is the maximum number of nodes on the root to leaf path. The height of a tree with a single node is considered as 1.
   This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^h-1$. This is a simple geometric series with h terms and sum of this series is $2^h - 1$.
   In some books, the height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$.
3. *In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$.* This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes $Log_2(N+1) - 1$.
4. *A Binary Tree with L leaves has at least ($Log_2L + 1$) levels.* A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

   ```
   L   <=  2l-1  [From Point 1]
   l =  Log2L + 1
   where l is the minimum number of levels.
   ```

5. *In a Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.*

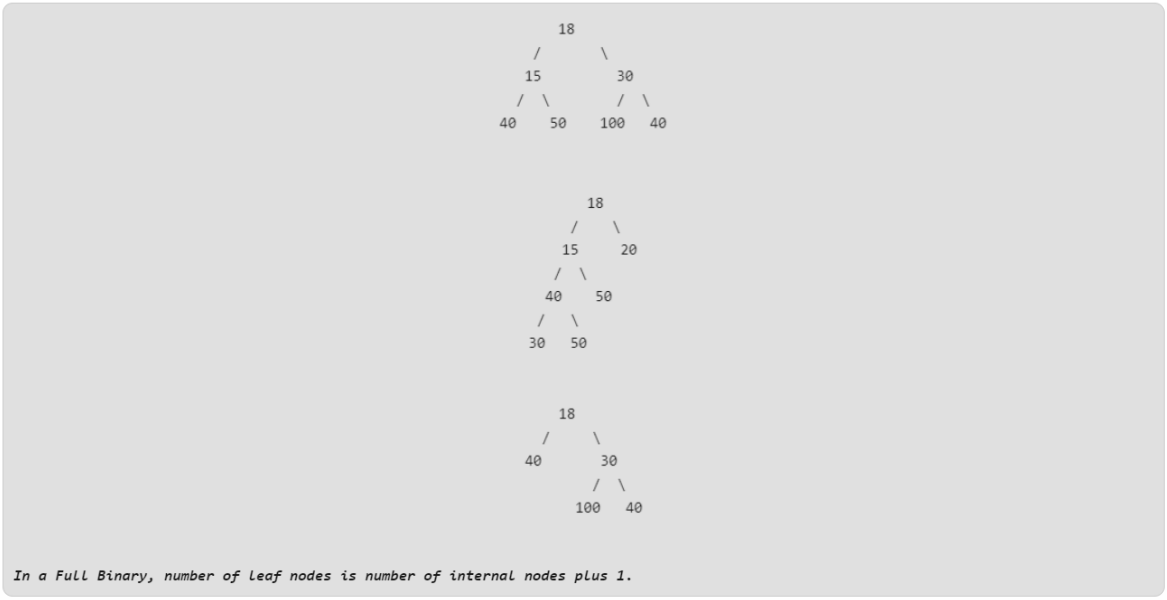
   ```
   L = T + 1
Where L = Number of leaf nodes
   T = Number of internal nodes with two children
   ```

▲

Types of Binary trees

Types of Binary Trees: Based on the structure and number of parents and children nodes, a Binary Tree is classified into the following common types:

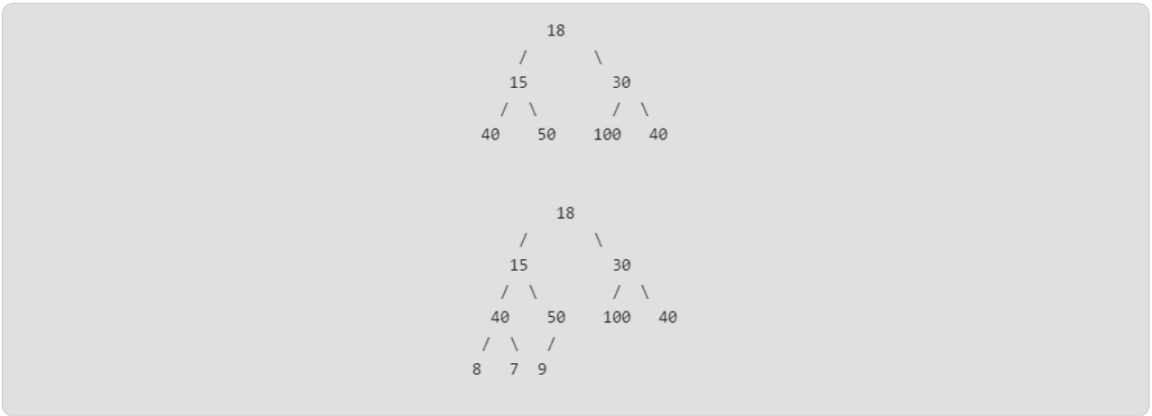
- **Full Binary Tree**: A Binary Tree is full if every node has either 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



In a Full Binary, number of leaf nodes is number of internal nodes plus 1.

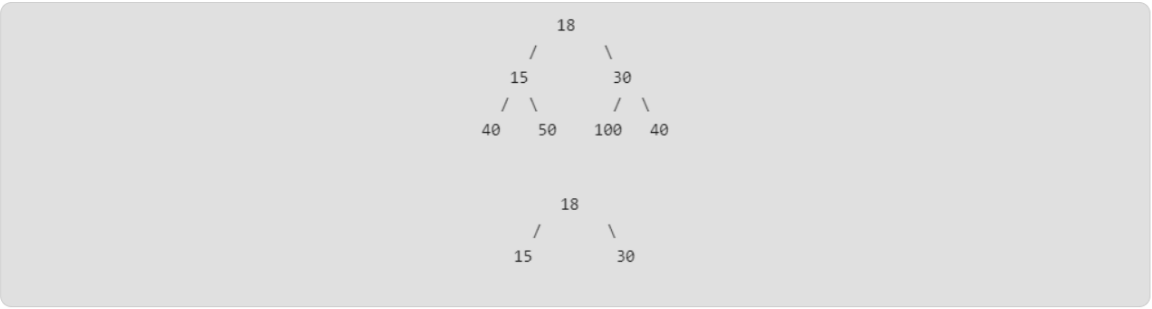
- **Complete Binary Tree**: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees:



- **Perfect Binary Tree**: A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.

Following are examples of Perfect Binary Trees:



A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ node.

Binary Tree Traversal

Preorder Traversal

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        Stack<TreeNode> st = new Stack<TreeNode>();
        ArrayList<Integer> res = new ArrayList<Integer>();
        while(root!=null || !st.isEmpty()){
            if(root!=null){
                st.push(root);
                res.add(root.val);
                root = root.left;
            }
            else{
                root = st.pop();
                root = root.right;
            }
        }
        return res;
    }
}
```

Inorder Traversal

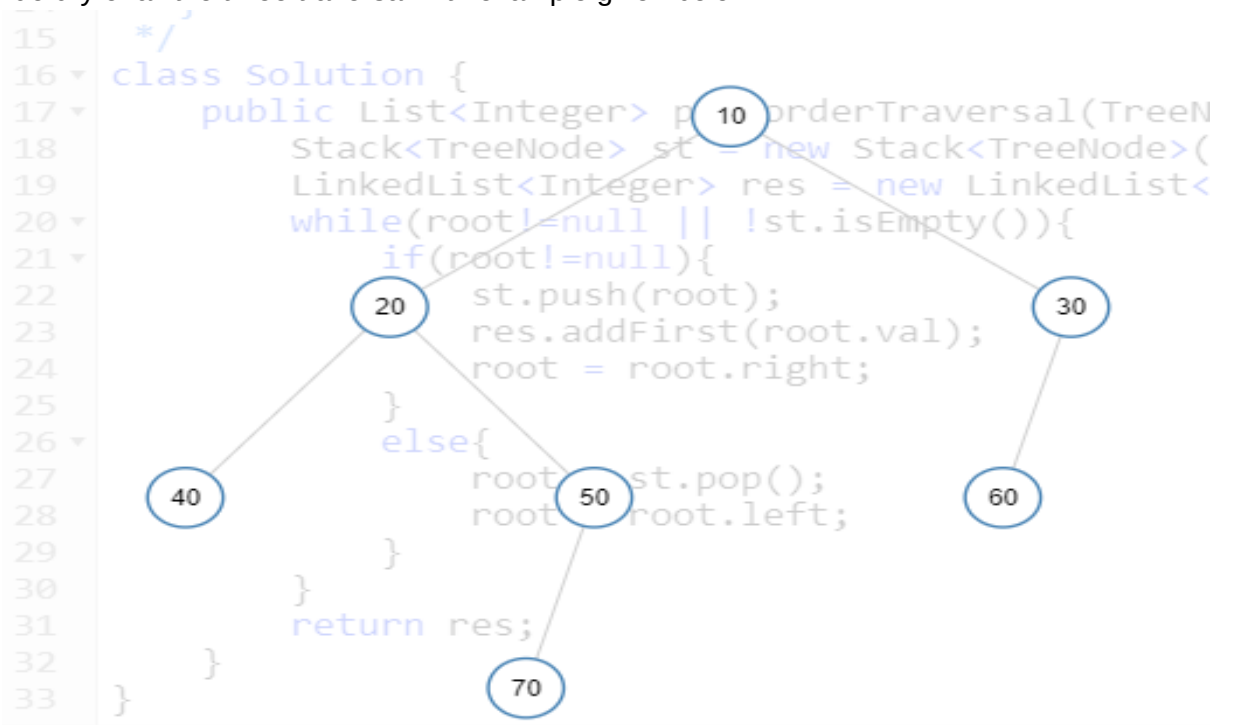
```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        Stack<TreeNode> st = new Stack<TreeNode>();
        ArrayList<Integer> res = new ArrayList<Integer>();
        while(root!=null || !st.isEmpty()){
            if(root!=null){
                st.push(root);
                root = root.left;
            }
            else{
                root = st.pop();
                res.add(root.val);
                root = root.right;
            }
        }
        return res;
    }
}
```

Postorder Traversal

Postorder traversal is reverse of preorder traversal

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        Stack<TreeNode> st = new Stack<TreeNode>();
        LinkedList<Integer> res = new LinkedList<Integer>();
        while(root!=null || !st.isEmpty()){
            if(root!=null){
                st.push(root);
                res.addFirst(root.val);
                root = root.right;
            }
            else{
                root = st.pop();
                root = root.left;
            }
        }
        return res;
    }
}
```

*do dry of all the three traversal with example given below



Threaded binary tree

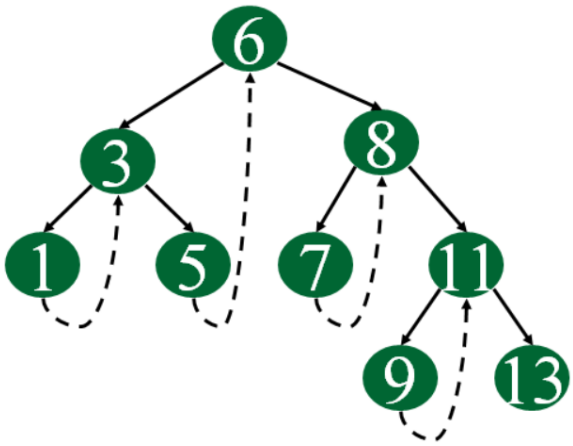
The *Inorder traversal of a Binary tree* can either be done using recursion or with the use of an auxiliary stack. **Threaded Binary Trees** are used to make the inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be *NULL* point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees:

1. ***Single Threaded:*** Where a NULL right pointers is made to point to the inorder successor (if successor exists).
2. ***Double Threaded:*** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

Note: The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Representation of a Threaded Node:

C++ Java

```java

class Node
    {
        int data;
        Node left, right;
        boolean rightThread;

        public Node(int item)
        {
            data = item;
            left = right = null;
        }
    }

```

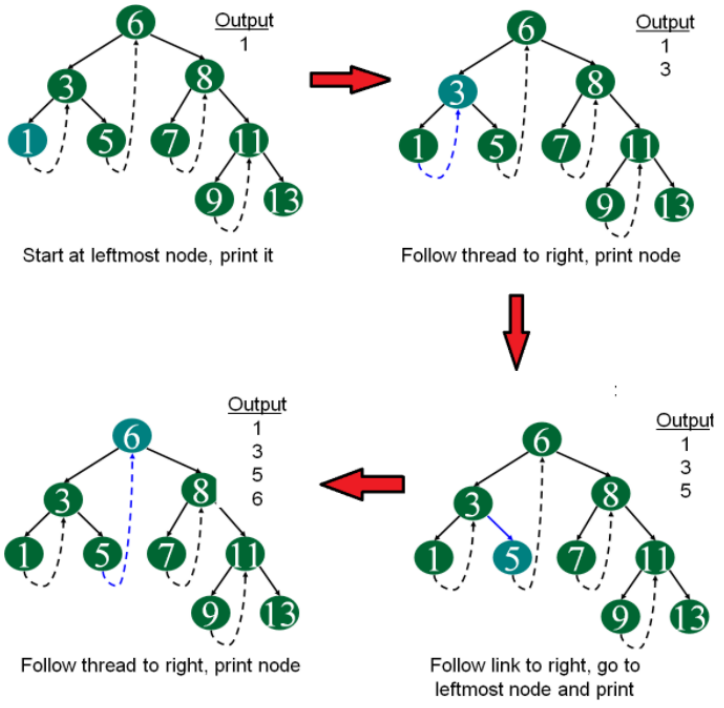
Since the right pointer in a Threaded Binary Tree is used for two purposes, the boolean variable *rightThread* is used to indicate whether the right pointer points to a right child or an inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

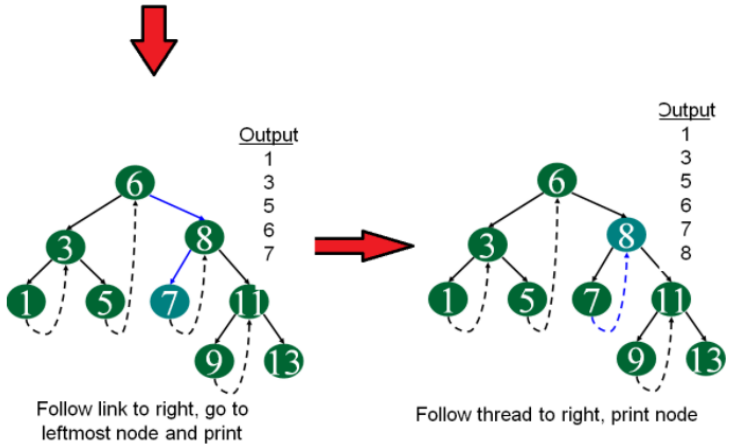
Inorder Traversal in a Threaded Binary Tree: Below is the algorithm to perform inorder traversal in a Threaded Binary Tree using threads:

1. Start from the root node, go to the leftmost node and print the node.
2. Check if there is a thread towards the right for the current node.
    - If Yes, then follow the thread to the node and print the data of node linked with this thread.
    - Otherwise follow the link to the right subtree, find the leftmost node in the right subtree and print the leftmost node.

   Repeat the above process until the complete tree is traversed.

Following diagram demonstrates the inorder traversal in a Threaded Binary Tree:





continue same way for remaining node.....

Code:

```
Node* leftMost(Node *N)
{
    if (N == NULL)
        return NULL;

    while (N->left != NULL)
        N = N->left;

    return N;
}

// Function to do inorder traversal in a
// threaded binary tree
void inOrder(Node *root)
{
    // Find leftmost node of the root node
    Node *cur = leftmost(root);

    // Until the complete tree is traversed
    while (cur != NULL)
    {
        print cur->data;

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
```

Problems on traversal

- Height of tree (Two ways: top-down, bottom-up)
- Determine two trees are identical: (Two problems: GfG, leetcode)
- Nodes at K distance from root: (GfG)

Level Order Traversal

Approach 1:

1. Find height of the binary tree
2. for i=0 to hrightOfTree
3. print all the nodes at distance i

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        TreeNode head = root;
        int h = heightOfTree(root, 0);
        ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();
        for(int i=0;i<h;i++){
            root = head;
            ArrayList<Integer> curr = new ArrayList<Integer>();
            nodesAtKdistance(root, curr, i);
            res.add(curr);
        }
        return res;
    }
    int heightOfTree(TreeNode root, int depth){
        if(root==null)
            return 0;
        return Math.max(heightOfTree(root.left, depth+1), heightOfTree(root.right, depth+1))+1;
    }
    void nodesAtKdistance(TreeNode root, ArrayList<Integer> curr, int k){
        if(root == null)
            return;
        if(k==0){
            curr.add(root.val);
            return;
        }
        nodesAtKdistance(root.left, curr, k-1);
        nodesAtKdistance(root.right, curr, k-1);
    }
}
```

Approach 2: Efficient

Use queue data structure. Queue holds every node at the next level.

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        if(root==null)
            return new ArrayList<List<Integer>>();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();
        q.add(root);
        while(!q.isEmpty()){
            int n = q.size();
            ArrayList<Integer> level = new ArrayList<Integer>();
            while(n-->0){
                TreeNode curr = q.remove();
                level.add(curr.val);
                if(curr.left!=null)
                    q.add(curr.left);
                if(curr.right!=null)
                    q.add(curr.right);
            }
            res.add(level);
        }
        return res;
    }
}
```

Time complexity: O(n)

Auxiliary space: Big O(n), big O(n) because in a queue we will have at most one level in every iteration.

It requires O(1) space when the binary tree has only one node at every level.

And O(n) when it has a different number of nodes at every level. In a perfect binary tree we have (l+1)/2 nodes in every level.

Another way of same approach:

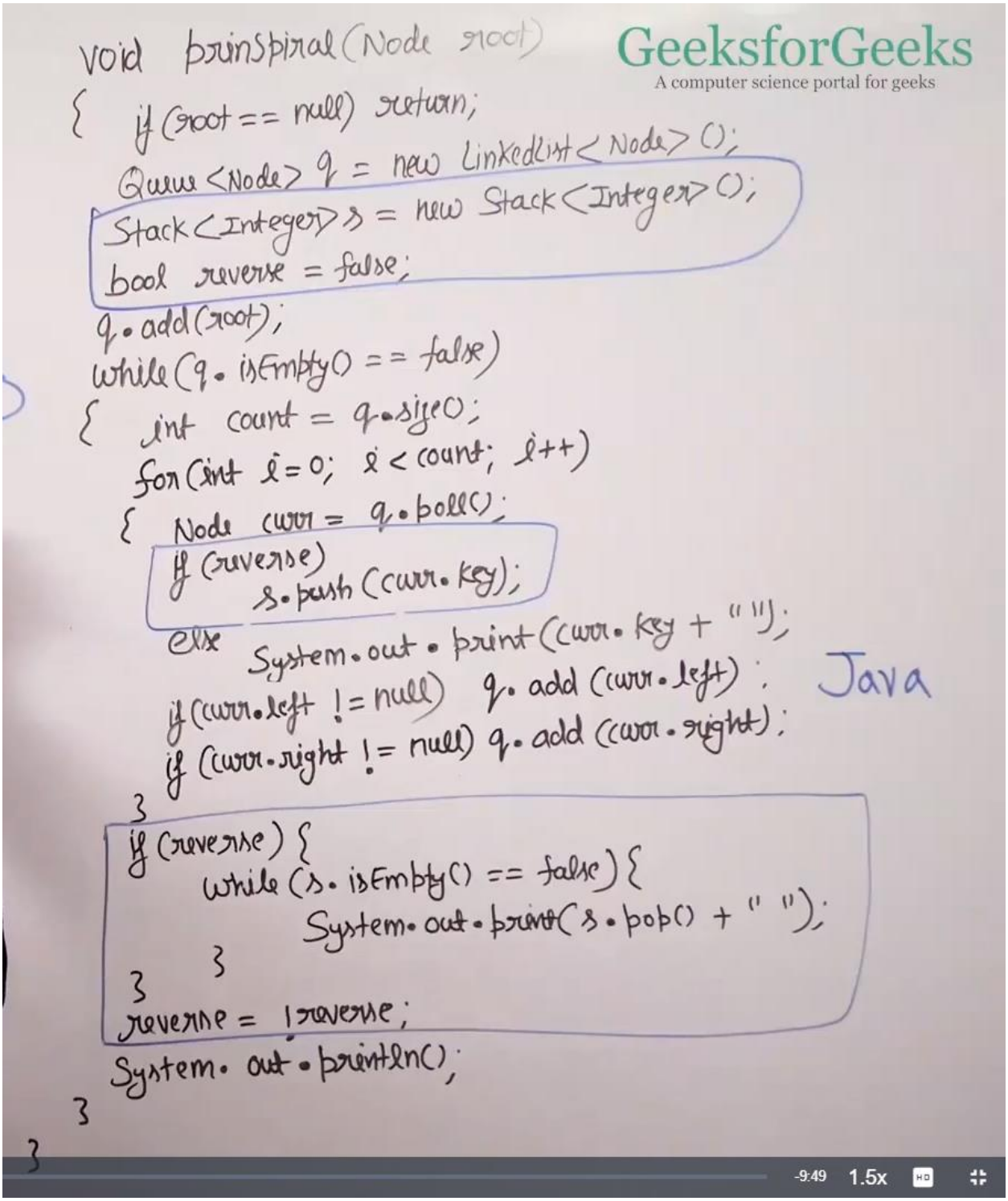
```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        if(root==null)
            return new ArrayList<List<Integer>>();
        Queue<TreeNode> q = new LinkedList<TreeNode>();
        ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();
        q.add(root);
        q.add(null);
        ArrayList<Integer> level = new ArrayList<Integer>();
        while(q.size()>1){
            TreeNode curr = q.remove();
            if(curr == null){
                res.add(level);
                level = new ArrayList<Integer>();
                q.add(null);
            }
            else{
                level.add(curr.val);
                if(curr.left!=null)
                    q.add(curr.left);
                if(curr.right!=null)
                    q.add(curr.right);
            }
        }
        res.add(level);
        return res;
    }
}
```

Here time complexity is Big O(n+h) where h is the height of the binary tree. We are adding and dequeuing null at every level.

Problems on Level Order Traversal

1. Level order traversal in spiral form: (GfG)

Method 1:



Method 2 (Efficient): use two stacks.

```
class Spiral
{
    ArrayList<Integer> findSpiral(Node node)
    {
       // Your code here
       if(node==null)
            return new ArrayList<Integer>();
       Stack<Node> st1 = new Stack<Node>();
       Stack<Node> st2 = new Stack<Node>();
       st1.push(node);
       ArrayList<Integer> res = new ArrayList<Integer>();
       while(!st1.isEmpty() || !st2.isEmpty()){
           if(!st1.isEmpty()){
               while(!st1.isEmpty()){
                   Node curr = st1.pop();
                   res.add(curr.data);
                   if(curr.right!=null)
                       st2.push(curr.right);
                   if(curr.left!=null)
                       st2.push(curr.left);
               }
           }
           else{
               while(!st2.isEmpty()){
                   Node curr = st2.pop();
                   res.add(curr.data);
                   if(curr.left!=null)
                       st1.push(curr.left);
                   if(curr.right!=null)
                       st1.push(curr.right);
               }
           }
       }
       return res;
    }
}
```

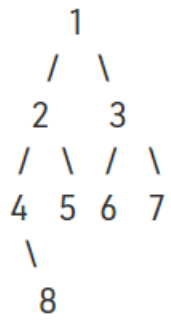
Method 3: Use dfs

```
printSpiral(tree)
  bool ltr = 0;
  for d = 1 to height(tree)
     printGivenLevel(tree, d, ltr);
     ltr ~= ltr /*flip ltr*/
```

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    if(ltr)
        printGivenLevel(tree->left, level-1, ltr);
        printGivenLevel(tree->right, level-1, ltr);
    else
        printGivenLevel(tree->right, level-1, ltr);
        printGivenLevel(tree->left, level-1, ltr);
```

2. Width of the binary tree: (GfG)

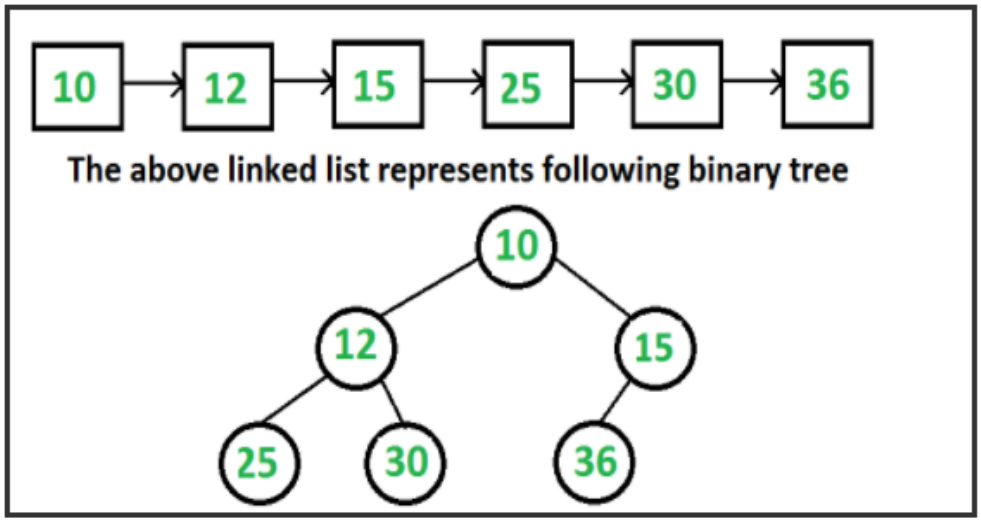
Given a Binary Tree, find the maximum width of it. **Maximum width** is defined as the maximum number of nodes in any level.
For example, maximum width of following tree is 4 as there are 4 nodes at 3rd level.



```
125 class Tree
126 {
127     // /* Function to get the maximum width of a binary tree*/
128     int getMaxWidth(Node root)
129     {
130         // Your code here
131         Queue<Node> q = new LinkedList<Node>();
132        int res = 0;
133        q.add(root);
134        while(!q.isEmpty()){
135            int n = q.size();
136            res = Math.max(res, n);
137            while(n-- > 0){
138                Node curr = q.remove();
139                if(curr.left!=null)
140                 q.add(curr.left);
141                if(curr.right!=null)
142                 q.add(curr.right);
143            }
144        }
145        return res;
146     }
147 }
148
```

3. Make Binary Tree From Linked List

Given a Linked List Representation of Complete Binary Tree. The task is to construct the Binary tree.
Note : The complete binary tree is represented as a linked list in a way where if root node is stored at position i, its left, and right children are stored at position **2*i+1**, **2*i+2** respectively.

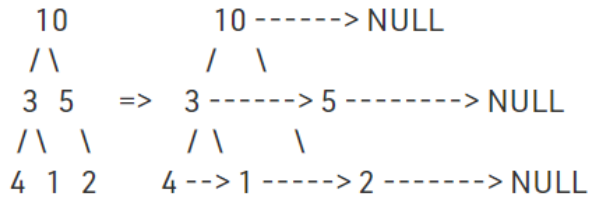


```
class GfG {
    public static Tree convert(Node head, Tree node) {
        // add code here.
        if(head==null)
            return node;
        Queue<Tree> q = new LinkedList<Tree>();
        Tree root = new Tree(head.data);
        q.add(root);
        head = head.next;
        while(head!=null){
            int n = q.size();
            while(n-- > 0 && head!=null){
                Tree curr = q.remove();
                curr.left = new Tree(head.data);
                head = head.next;
                q.add(curr.left);
                if(head!=null){
                    curr.right = new Tree(head.data);
                    head = head.next;
                    q.add(curr.right);
                }
            }
        }
        return root;
    }
```

4. Connect Nodes at Same Level

Given a binary tree, connect the nodes that are at same level. You'll be given an addition **nextRight** pointer for the same.

Initially, all the **nextRight** pointers point to **garbage** values. **Your function** should set these pointers to point next right for each node.



```
140
141 class Tree
142 {
143     /*
144     Node class is Defined as follows:
145     class Node{
146         int data;
147         Node left;
148         Node right;
149         Node nextRight;
150         Node(int data){
151             this.data = data;
152             left=null;
153             right=null;
154             nextRight = null;
155         }
156     }
157     */
158     public static void connect(Node p) {
159         // code here.
160         Queue<Node> q = new LinkedList<Node>();
161         q.add(p);
162         while(!q.isEmpty()){
163             int n = q.size();
164             Node  prev = null;
165             while(n-- > 0){
166                 Node curr = q.remove();
167                 if(curr.left!=null)
168                     q.add(curr.left);
169                 if(curr.right!=null)
170                     q.add(curr.right);
171                 if(prev!=null)
172                     prev.nextRight = curr;
173                 prev = curr;
174             }
175             if(prev!=null)
176                 prev.nextRight = null;
177         }
178     }
179 }
```

5. Populating Next Right Pointers in Each Node (leetcode)

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Follow up:

- You may only use constant extra space.
- Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

Example 1:

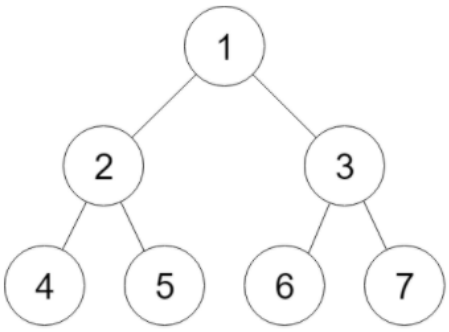


Figure A

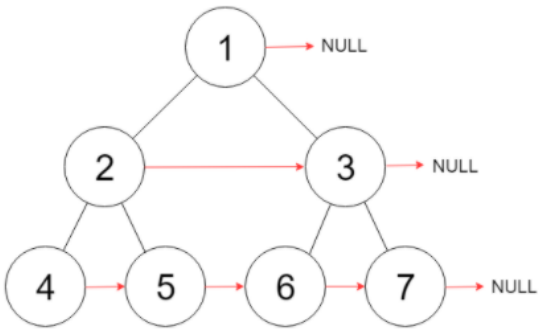


Figure B

```
Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate each
next pointer to point to its next right node, just like in Figure B. The serialized output is
in level order as connected by the next pointers, with '#' signifying the end of each level.
```

```
class Solution {
    public Node connect(Node root) {
        if(root==null)
            return root;
        Node level = root;
        while(level.left!=null){
            Node curr = level;
            while(curr!=null){
                curr.left.next = curr.right;
                if(curr.next!=null)
                    curr.right.next = curr.next.left;
                curr = curr.next;
            }
            level = level.left;
        }
        return root;
    }
}
```

This solution requires O(n) extra space. Below solution does not need any extra space.

```
class Solution {
    public Node connect(Node root) {
        Node level = root;
        while(level!=null){
            Node curr = level;
            while(curr!=null && curr.left!=null){
                if(curr.left!=null)
                    curr.left.next = curr.right;
                if(curr.next!=null)
                    curr.right.next = curr.next.left;
                else
                    curr.right.next = null;
                curr = curr.next;
            }
            level = level.left;
        }
        return root;
    }
}
```

6. Populating Next Right Pointers in Each Node II

Given a binary tree

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Follow up:

- You may only use constant extra space.
- Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

Example 1:

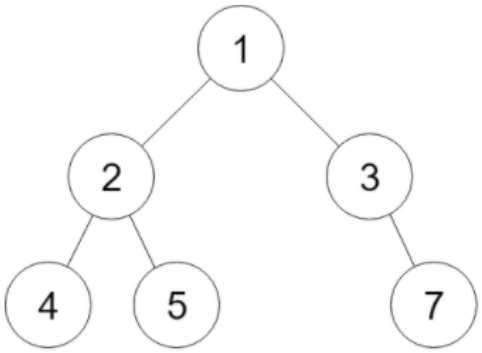


Figure A

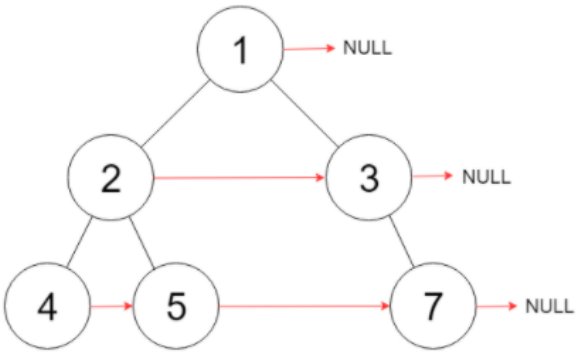


Figure B

```
Input: root = [1,2,3,4,5,null,7]
Output: [1,#,2,3,#,4,5,7,#]
Explanation: Given the above binary tree (Figure A), your function should populate each next
pointer to point to its next right node, just like in Figure B. The serialized output is in
level order as connected by the next pointers, with '#' signifying the end of each level.
```

```java
class Solution {
    public Node connect(Node root) {
        Node level = root;
        while(level!=null){
            Node curr = level;
            while(curr!=null){
                if(curr.left!=null){
                    if(curr.right!=null)
                        curr.left.next = curr.right;
                    else {
                        Node tmp = curr.next;
                        while(tmp!=null && (tmp.left==null && tmp.right==null))
                            tmp = tmp.next;
                        if(tmp!=null){
                            if(tmp.left!=null)
                                curr.left.next = tmp.left;
                            else
                                curr.left.next = tmp.right;
                        }
                    }
                }
                if(curr.right!=null){
                    Node tmp = curr.next;
                    while(tmp!=null && (tmp.left==null && tmp.right==null))
                        tmp = tmp.next;
                    if(tmp!=null){
                        if(tmp.left!=null)
                            curr.right.next = tmp.left;
                        else
                            curr.right.next = tmp.right;
                    }
                }
                curr = curr.next;
            }
            if(level.left!=null)
                level = level.left;
            else if(level.right!=null)
                level = level.right;
              else{
                  Node tmp = level.next;
                  while(tmp!=null && (tmp.left==null && tmp.right==null))
                          tmp = tmp.next;
                  if(tmp!=null){
                      if(tmp.left!=null)
                          level = tmp.left;
                      else
                          level = tmp.right;
                  }
                  else
                      level = null;

              }

          }
          return root;
      }
}
```

Simple solution

The idea is simple: level-order traversal.
You can see the following code:

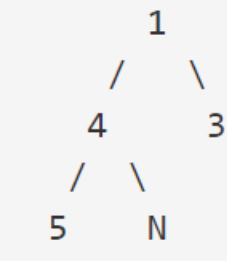
```
public class Solution {
    public void connect(TreeLinkNode root) {

        while(root != null){
            TreeLinkNode tempChild = new TreeLinkNode(0);
            TreeLinkNode currentChild = tempChild;
            while(root!=null){
                if(root.left != null) { currentChild.next = root.left; currentChild = currentChild.next;}
                if(root.right != null) { currentChild.next = root.right; currentChild = currentChild.next;}
                root = root.next;
            }
            root = tempChild.next;
        }
    }
}
```

Other Problems:

1. Children Sum Parent

Given a Binary Tree. Check whether all of its nodes have the value equal to the sum of their child nodes.

Input:



Output: 0

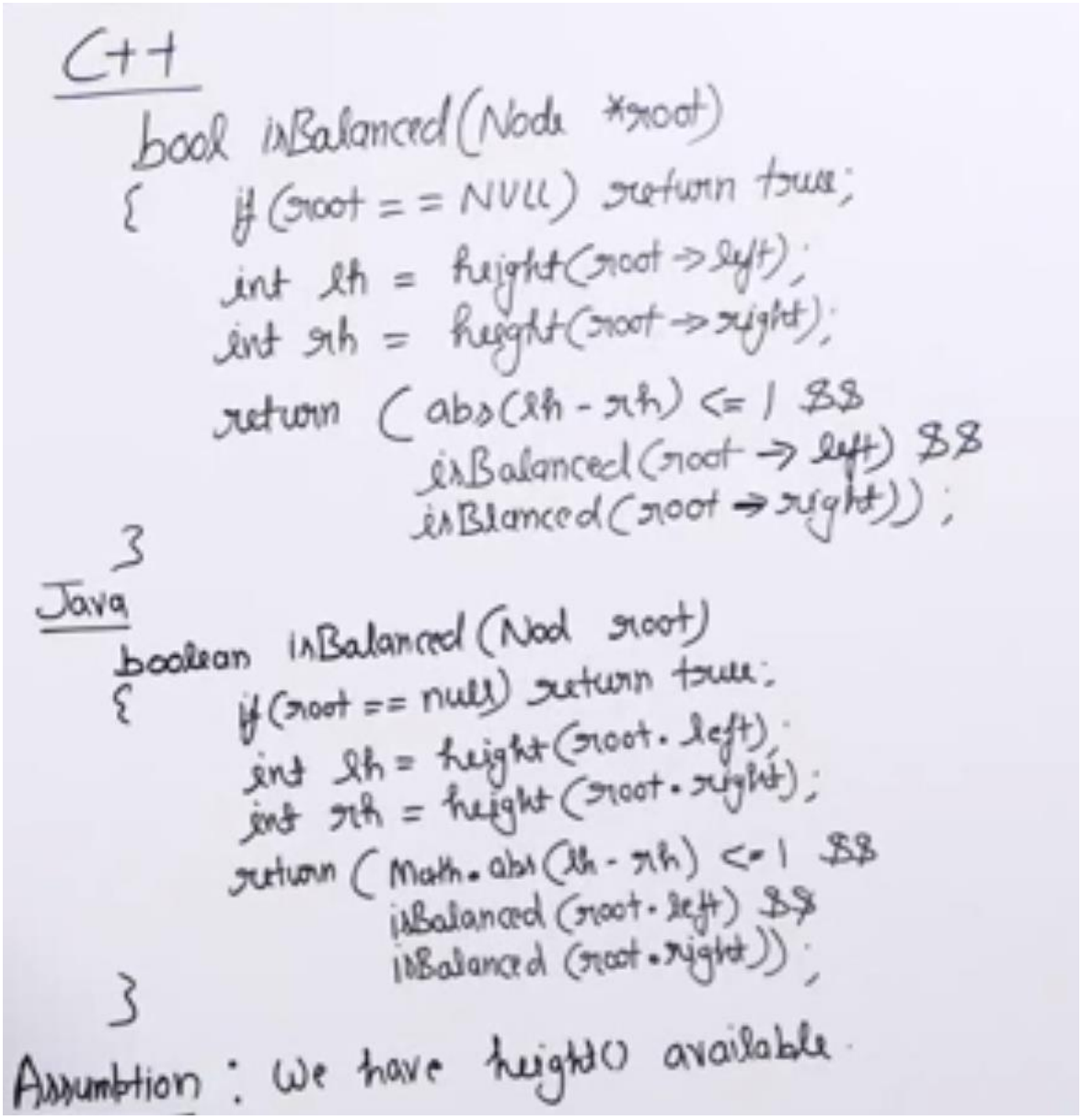
Explanation: Here, 1 is the root node and 4, 3 are its child nodes. 4 + 3 = 7 which is not equal to the value of root node. Hence, this tree does not satisfy the given conditions.

```
116 int isSumProperty(Node *root)
117 {
118  // Add your code here
119  if(root == NULL)
120     return 1;
121  if(!root->left && !root->right)
122     return 1;
123  int left = root->left!=NULL ? root->left->data : 0;
124  int right = root->right!=NULL ? root->right->data : 0;
125  return left+right == root->data && isSumProperty(root->left) && isSumProperty(root->right);
126 }
```

2. Check for Balanced tree

Given a binary tree, find if it is height balanced or not.
A tree is height balanced if the difference between heights of left and right subtrees is not more than one for all nodes of the tree.

Naive solution:



Time complexity: O(n^2)
Because we are finding the height of every subtree.

Efficient solution: Every child will give the height of its subtree. Using this fact we can determine if the tree is balanced or not. If the left height and right height difference>1 the return -1 else tree is balanced and return height of the tree.

```
129 class Tree
130 {
131
132     /* This function should return tree if passed  tree
133     is balanced, else false. */
134     boolean isBalanced(Node root)
135     {
136         // Your code here
137         return height(root) >= 0;
138     }
139     int height(Node root){
140         if(root==null)
141             return 0;
142
143         int leftheight = height(root.left);
144         if(leftheight==-1)
145             return -1;
146         int rightheight = height(root.right);
147         if(rightheight==-1)
148             return -1;
149
150         if(Math.abs(leftheight-rightheight)>1)
151             return -1;
152         else
153             return Math.max(leftheight, rightheight)+1;
154     }
155 }
```

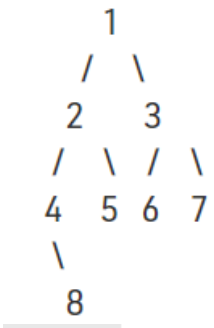
Other solution:

```
class Tree
{

     /* This function should return tree if passed  tree
     is balanced, else false. */
    boolean isBalanced;
    boolean isBalanced(Node root)
    {
    // Your code here
        isBalanced = true;
        height(root);
        return isBalanced;
    }
    int height(Node root){
        if(!isBalanced){
            return 0;
        }
        if(root==null)
            return 0;
        int left = height(root.left)+1;
        int right = height(root.right)+1;
        int diff = Math.abs(left-right);
        if(diff>1)
            isBalanced = false;
        return Math.max(left, right);
    }
}
```

3. Left view of binary tree

Given a Binary Tree, print Left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from Left side. The task is to complete the function **leftView()**, which accepts root of the tree as argument.

Left view of following tree is 1 2 4 8.

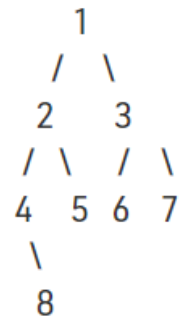



```
122 class Tree
123 {
124     ArrayList<Integer> leftView(Node root)
125     {
126       // Your code here
127       ArrayList<Integer> res = new ArrayList<Integer>();
128       leftView(root, res);
129       return res;
130     }
131     void leftView(Node root, ArrayList<Integer> res){
132         if(root==null)
133             return;
134         res.add(root.data);
135         if(root.left!=null)
136             leftView(root.left, res);
137         else
138             leftView(root.right, res);
139     }
140 }
```

4. Right view of binary tree

Given a Binary Tree, find **Right view** of it. Right view of a Binary Tree is set of nodes visible when tree is viewed from **right** side.

Right view of following tree is 1 3 7 8.




```
132 class Tree{
133     ArrayList<Integer> rightView(Node root) {
134         //add code here.
135         if(root==null)
136             return new ArrayList<Integer>();
137         Queue<Node > q = new LinkedList<Node>();
138         ArrayList<Integer> res = new ArrayList<Integer>();
139         q.add(root);
140         while(!q.isEmpty()){
141             int n = q.size();
142             while(n-- > 0){
143                 Node curr = q.remove();
144                 if(n==0)
145                     res.add(curr.data);
146                 if(curr.left!=null)
147                     q.add(curr.left);
148                 if(curr.right!=null){
149                     q.add(curr.right);
150                 }
151             }
152         }
153         return res;
154     }
155 }
```

Variations of left and right view of Binary tree: Top and Bottom view

- **Top View**: Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. A node **x** is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to the horizontal distance of x minus 1, and that of right child is the horizontal distance of x plus 1.

  So, the idea is to do a level order traversal of the tree and calculate the horizontal distance of every node from the root node and print those nodes which are the first nodes of a particular horizontal distance.

  Hashing can be used to keep a check on whether any node with a particular horizontal distance is encountered yet or not.

```
void topview(Node* root)
{
    if(root==NULL)
       return;
    queue < Node* > q
    map < int,int > m
    int hd=0
    root->hd=hd

     // push node and horizontal distance to queue
    q.push(root);
    while(!q.empty())
    {
        hd=root->hd
        // check whether node at hd distance seen or not
        if(m.find(hd)==false)
            m[hd]=root->data
        if(root->left)
        {
            root->left->hd=hd-1
            q.push(root->left)
        }
        if(root->right)
        {
            root->right->hd=hd+1
            q.push(root->right)
        }
        q.pop()
        root=q.front()
    }
    for(it=m.begin();it!=m.end();it++)
    {
        print(it->second)
    }
}
```

- **Bottom View**: The Bottom View of a binary tree can be printed using the similar approach as that of printing the Top View. A node **x** is there in output if x is the bottom most instead of the top most node at its horizontal distance.

  The process of printing the bottom view is almost the same as that of top view with a little modification that while storing the node's data along with a particular horizontal distance in the map, keep updating the node's data in the map for a particular horizontal distance so that the map contains the last node appearing with a particular horizontal distance instead of first.

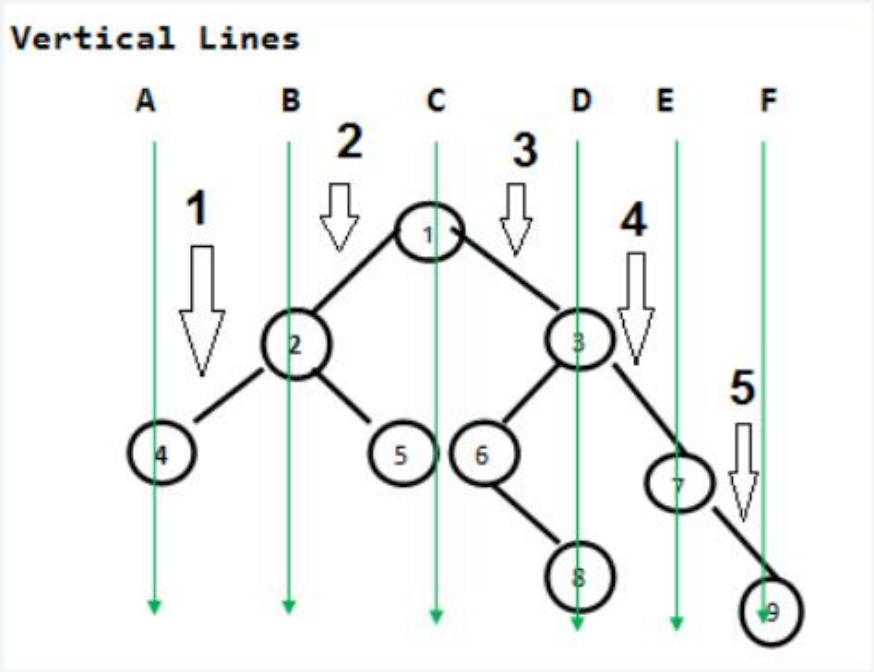
5. Vertical width of binary tree

Input:

```
          1
       /    \
      2      3
     / \    / \
    4   5  6   7
            \   \
             8   9
```

Output: 6

Explanation: The width of a binary tree is the number of vertical paths in that tree.

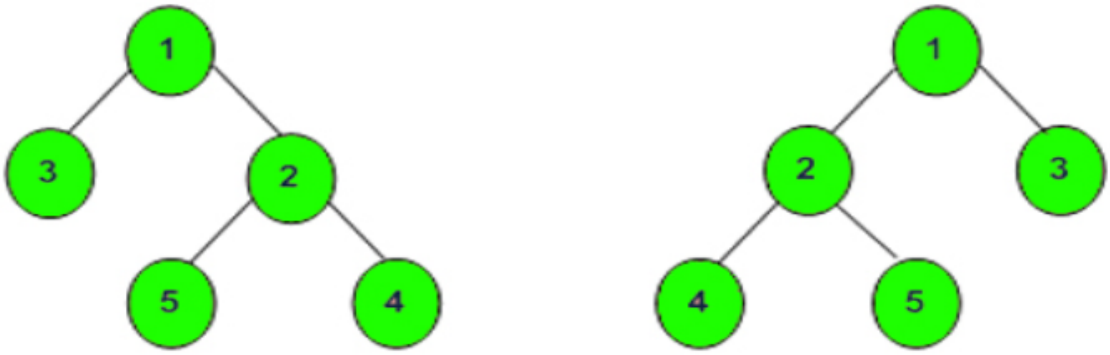


The above tree contains 6 vertical lines.

```
class Tree
{
    static int leftMost, rightMost;
    public static int verticalWidth(Node root)
    {
        // code here.
        if(root == null)
            return 0;
        leftMost = 0;
        rightMost = 0;
        width(root, 0);
        return (rightMost-leftMost)+1;
    }
    static void width(Node root, int width){
        if(root==null)
            return ;
       width(root.left, width-1);
       width(root.right, width+1);
       leftMost = Math.min(width, leftMost);
       rightMost = Math.max(width, rightMost);
    }
}
```

6. Mirror tree

Given a Binary Tree, convert it into its mirror.



Mirror Trees

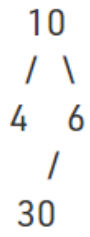
```
131 class Tree
132 {
133     void mirror(Node node)
134     {
135         // Your code here
136         doMirror(node);
137     }
138     Node doMirror(Node root){
139         if(root==null)
140             return root;
141         Node temp = doMirror(root.left);
142         root.left= doMirror(root.right);
143         root.right = temp;
144         return root;
145     }
146 }
```

7. Check if subtree

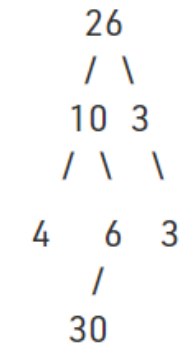
Given two binary trees with head reference as **T** and **S** having at most **N** nodes. The task is to check if S is present as subtree in T.
A subtree of a tree T1 is a tree T2 consisting of a node in T1 and all of its descendants in T1.

Example:

S:



T:

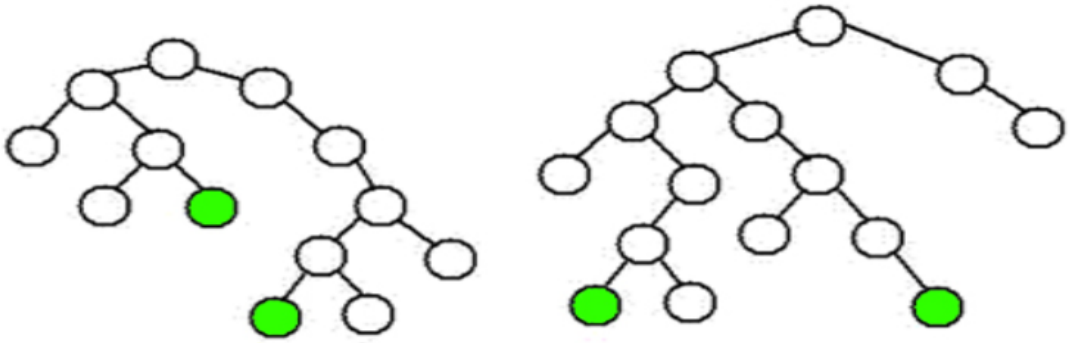


In above example S is subtree of T.

```
class Tree {
    public static boolean isSubtree(Node T, Node S) {
        // add code here.
        if(T==null)
            return false;
        if(T.data==S.data && isSame(T, S))
            return true;
        return isSubtree(T.left, S) || isSubtree(T.right, S);
    }
    static boolean isSame(Node T, Node S){
        if(T==null && S==null)
            return true;
        if((T==null && S!=null) || (T!=null && S==null))
            return false;
        if(T.data!=S.data)
            return false;
        return isSame(T.left, S.left) && isSame(T.right, S.right);
    }
}
```

8. Diameter of Binary Tree

Given a Binary Tree, **find diameter of it**.
The diameter of a tree is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



Diameter, 9 nodes, through root

Diameter, 9 nodes, NOT through root

Solution: The diameter of a tree T is the largest of the following quantities:

- The diameter of T's left subtree.
- The diameter of T's right subtree.
- The longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T).

The longest path between leaves that goes through a particular node say, ***nd*** can be calculated as:

> *1 + height of left subtree of nd + height of right subtree of nd*

Therefore, final **Diameter** of a node can be calculated as:

> *Diameter = maximum(lDiameter, rDiameter, 1 + lHeight + rHeight)*
> Where,
> **lDiameter** = Diameter of left subtree
> **rDiameter** = Diameter of right subtree
> **lHeight** = Height of left subtree
> **rHeight** = Height of right subtree

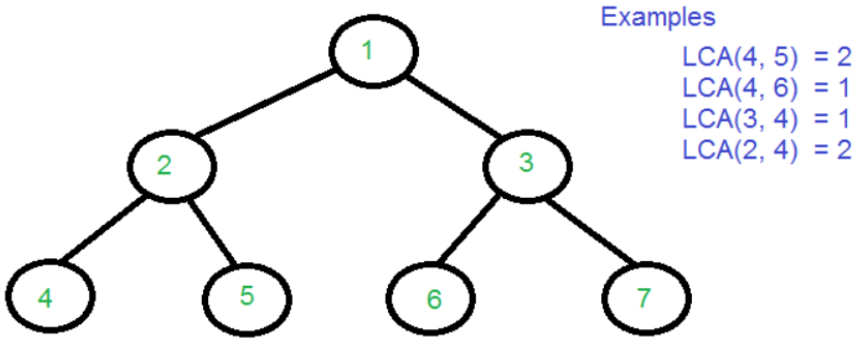
```
106 class Tree {
107     /* Complete the function to get diameter of a binary tree */
108     int max;
109     int diameter(Node root) {
110         // Your code here
111         max = 0;
112         height(root);
113         return max;
114     }
115     int height(Node root){
116         if(root==null)
117             return 0;
118         int left = height(root.left);
119         int right = height(root.right);
120         max = Math.max(max, left+right+1);
121         return Math.max(left, right)+1;
122     }
123 }
124
```

9. Lowest Common Ancestor in a Binary Tree

Given a **Binary Tree** and the value of two nodes **n1** and **n2**. The task is to find the *lowest common ancestor* of the nodes n1 and n2 in the given Binary Tree.

> *The **LCA** or **Lowest Common Ancestor** of any two nodes N1 and N2 is defined as the common ancestor of both the nodes which is closest to them. That is the distance of the common ancestor from the nodes N1 and N2 should be least possible.*

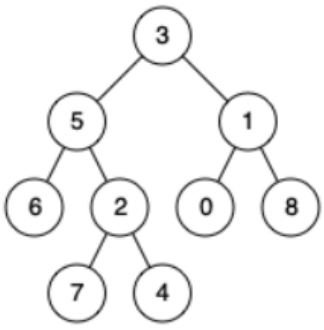
Below image represents a tree and LCA of different pair of nodes (N1, N2) in it:



Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Finding LCA

Method 1: The simplest method of finding LCA of two nodes in a Binary Tree is to observe that the LCA of the given nodes will be the last common node in the paths from the root node to the given nodes.

For Example: consider the above-given tree and nodes **4** and **5**.

- Path from root to node 4: [1, 2, 4]
- Path from root to node 5: [1, 2, 5].

*The last common node is **2** which will be the LCA.*

Algorithm:

1. Find the path from the **root** node to node **n1** and store it in a vector or array.
2. Find the path from the **root** node to node **n2** and store it in another vector or array.
3. Traverse both paths untill the values in arrays are same. Return the common element just before the mismatch.

Find path from root to both the given nodes. Then find the common nodes through the both the paths.

```
10 class Solution {
11     public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
12         ArrayList<TreeNode> path1 = new ArrayList<TreeNode>();
13         ArrayList<TreeNode> path2 = new ArrayList<TreeNode>();
14         if(!findPath(root, p, path1) || !findPath(root, q, path2))
15             return root;
16         int i=0;
17         for(i=0;i<path1.size()-1&&i<path2.size()-1;i++){
18             if(path1.get(i+1)!=path2.get(i+1))
19                 return path1.get(i);
20         }
21         if(i==path1.size()-1)
22             return path1.get(path1.size()-1);
23         if(i==path2.size()-1)
24             return path2.get(path2.size()-1);
25         return root;
26     }
27     boolean findPath(TreeNode root, TreeNode n, ArrayList<TreeNode> path){
28         if(root==null)
29             return false;
30         path.add(root);
31         if(root==n)
32             return true;
33        if(findPath(root.left, n, path) || findPath(root.right, n, path))
34             return true;
35        path.remove(path.size()-1);
36        return false;
37     }
38 }
```

Analysis: *The **time complexity** of the above solution is O(N) where N is the number of nodes in the given Tree and the above solution also takes O(N) **extra space**.*

Method 2:

Method 2: The method 1 finds LCA in O(N) time but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.

The idea is to traverse the tree starting from the root node. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtrees. The node which has one key present in its left subtree and the other key present in the right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise, LCA lies in the right subtree.

Under assumption that both the given nodes present in given root.

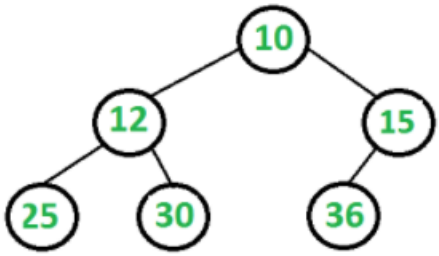
There are conditions that can occur:

i. root is one of the given nodes
ii. left or right subtree of the given node contains one of the given nodes
iii. left subtree contains one given node and right subtree contains another given node
iv. neither left nor right subtree contains any of the given node

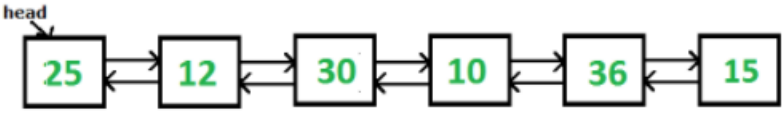
```
 */
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null)
            return root;
        if(root==p || root==q)
            return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left!=null && right!=null)
            return root;
        if(left!=null)
            return left;
        if(right!=null)
            return right;
        return null;
    }
}
```

10. Binary Tree to DLL

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (leftmost node in BT) must be the head node of the DLL.



The above tree should be in-place converted to following Doubly Linked List(DLL).




```
class GfG
{
    Node bToDLL(Node root){
        Node tail = construct(root);
        Node head = tail;
        while(head.left!=null)
            head = head.left;
       return head;
    }
    Node construct(Node root){
        if(root==null)
            return null;
        Node prev = construct(root.left);
        if(prev!=null){
            while(prev.right!=null)
                prev = prev.right;
            prev.right = root;
        }
        root.left = prev;
        Node next = construct(root.right);
        if(next!=null){
            while(next.left!=null)
                next = next.left;
            next.left = root;
        }
        root.right = next;
        return root;
    }
}
```

11. Binary Tree to CDLL

Given a Binary Tree of **N** edges. The task is to convert this to a Circular Doubly Linked List(**CDLL**) in-place. The left and right pointers in nodes are to be used as previous and next pointers respectively in converted CDLL. The order of nodes in CDLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the CDLL.

Example 1:

```
Input:
      1
    /   \
   3     2
Output:
3 1 2
2 1 3
Explanation: After converting tree to CDLL
when CDLL is is traversed from head to
tail and then tail to head, elements
are displayed as in the output.
```

```
148 class Tree
149 {
150
151     Node bTreeToClist(Node root){
152         Node tail = construct(root);
153         Node head = tail;
154         while(head.left!=null)
155             head = head.left;
156        while(tail.right!=null)
157             tail = tail.right;
158        tail.right = head;
159        head.left = tail;
160        return head;
161     }
162     Node construct(Node root){
163         if(root==null)
164             return null;
165         Node curr = new Node(root.data);
166         Node prev = construct(root.left);
167         if(prev!=null){
168             while(prev.right!=null)
169                 prev = prev.right;
170             prev.right = curr;
171         }
172         curr.left = prev;
173         Node next = construct(root.right);
174         if(next!=null){
175             while(next.left!=null)
176                 next = next.left;
177             next.left = curr;
178         }
179         curr.right = next;
180         return curr;
181     }
182 }
183
```

12. Construct Binary Tree from Parent Array

Given an array of size **N** that represents a Tree in such a way that array indexes are values in tree nodes and array values give the parent node of that particular index (or node). The value of the root node index would always be **-1** as there is no parent for root. Construct the standard linked representation of Binary Tree from this array representation.

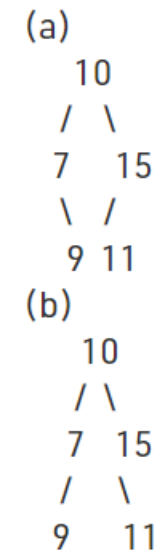
Example 1:

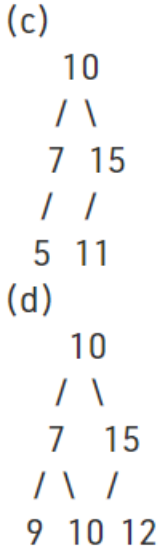
```
Input:
N = 7
parent[] = {-1,0,0,1,1,3,5}
Output: 0 1 2 3 4 5 6
Explanation:For the array parent[] = {-1,
0, 0, 1, 1, 3, 5}; the tree generated
will have a sturcture like
          0
        /   \
       1     2
     / \
    3   4
   /
  5
/
6
```

```
class GfG {
    public static Node createTree(int arr[], int n) {
        //Your code here
        HashMap<Integer, Node> hm = new HashMap<Integer, Node>();
        for(int i=0;i<=n;i++)
            hm.put(i, new Node(i));

        Node root = null;
        for(int i=0;i<n;i++)
            if(arr[i]==-1)
                root = hm.get(i);

        for(int i=0;i<n;i++){
            if(arr[i]==-1)
                continue;
            Node curr = hm.get(arr[i]);
            if(curr.left==null){
                curr.left = hm.get(i);
                hm.remove(i);
                hm.put(i, curr.left);
            }
            else{
                curr.right = hm.get(i);
                hm.remove(i);
                hm.put(i, curr.right);
            }
        }
        return root;
    }
}
```

13. Foldable Binary Tree

Given a binary tree, check if the tree **can be folded or not**. A tree can be folded if left and right subtrees of the tree are **structure wise same**. An empty tree is considered as foldable.
Consider the below trees:
(a) and (b) can be folded.
(c) and (d) cannot be folded.





```
class Tree {
    boolean IsFoldable(Node node)
    {
        // your code
       if(node==null)
            return true;
       return isSame(node.left, node.right);
    }
    boolean isSame(Node r1, Node r2){
        if(r1==null && r2==null)
            return true;
       if((r1==null && r2!=null) || (r1!=null && r2==null))
            return false;
       return isSame(r1.left, r2.right) && isSame(r1.right, r2.left);
    }
}
```

14. Maximum path sum from any node

Given a binary tree, the task is to find the maximum path sum. The path may start and end at any node in the tree.

Example 1:

```
Input:
     10
    /  \
   2   -25
  / \  /  \
 20 1  3  4
Output: 32
Explanation: Path in the given tree goes
like 10 , 2 , 20 which gives the max
sum as 32.
```

Hint 1 Full Solution

For each node there can be four ways that the max path goes through the node:
1. Node only
2. Max path through Left Child + Node
3. Max path through Right Child + Node
4. Max path through Left Child + Node + Max path through Right Child

The idea is to keep trace of four paths and pick up the max one in the end. An important thing to note is, root of every subtree need to return maximum path sum such that at most one child of root is involved. This is needed for parent function call.

```
class Tree
{
    // This function should returns sum of
    // maximum sum path from any node in
    // a tree rooted with given root.
    int  max;
    int findMaxSum(Node node) {
        //your code goes here
        max = Integer.MIN_VALUE;
        sum(node);
        return max;
    }
    int sum(Node root){
        if(root==null)
            return 0;
        int data = root.data;
        int left = sum(root.left);
        int right = sum(root.right);
        max = Math.max(max, Math.max(Math.max(data, left+data+right), Math.max(data+left, data+right)));
        return Math.max(data, Math.max(data+left, data+right));
    }
}

```

15. Maximum difference between node and its ancestor (GfG)

Given a Binary Tree, you need to find the maximum value which you can get by subtracting the value of node B from the value of node A, where A and B are two nodes of the binary tree and A is an ancestor of B.

Example 1:

```
Input:
    5
 /    \
2      1
Output: 4
Explanation:The maximum difference we can
get is 4, which is bewteen 5 and 1.
```

Example 2:

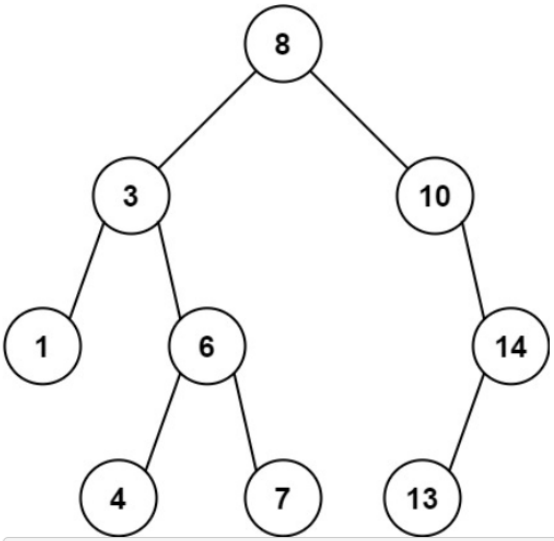
```
Input:
      1
    /    \
   2      3
           \
            7
Output: -1
Explanation:The maximum difference we can
get is -1, which is between 1 and 2.
```

```
125 class Tree
126 {
127     int max;
128     int maxDiff(Node root)
129     {
130         //your code here
131         max = Integer.MIN_VALUE;
132         maxDiff1(root);
133         return max;
134     }
135     int maxDiff1(Node root){
136         if(root==null)
137             return Integer.MAX_VALUE;
138         int left = maxDiff1(root.left);
139         int right = maxDiff1(root.right);
140         max = Math.max(max, Math.max(root.data-left, root.data-right));
141         return Math.min(root.data, Math.min(left, right));
142     }
143 }
144
145
```

16. Maximum Difference Between Node and Ancestor: (leetcode)

Given the `root` of a binary tree, find the maximum value `V` for which there exist **different** nodes `A` and `B` where `V = |A.val - B.val|` and `A` is an ancestor of `B`.

A node `A` is an ancestor of `B` if either: any child of `A` is equal to `B`, or any child of `A` is an ancestor of `B`.



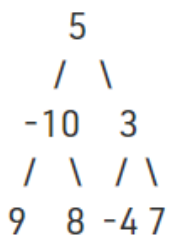
Input: root = [8,3,10,1,6,null,14,null,null,4,7,13]
Output: 7
Explanation: We have various ancestor-node differences, some of which are given below :
|8 - 3| = 5
|3 - 7| = 4
|8 - 1| = 7
|10 - 13| = 3
Among all possible differences, the maximum value of 7 is obtained by |8 - 1| = 7.

```java
class Solution {
    public int maxAncestorDiff(TreeNode root) {
        if(root==null)
            return 0;
        return helper(root, root.val, root.val);
    }
    int helper(TreeNode root, int min, int max){
        if(root==null)
            return max-min;
        min = Math.min(min, root.val);
        max = Math.max(max, root.val);
        int left = helper(root.left, min, max);
        int right = helper(root.right, min, max);
        return Math.max(left, right);
    }
}
```

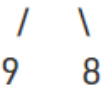
17. Count Number of SubTrees having given Sum

Given a binary tree and an integer **X**. Your task is to complete the function **countSubtreesWithSumX()** that returns the count of the number of subtress having total node's data sum equal to the value **X**.

Example: For the tree given below:



Subtree with sum 7:



and one node 7.

```
125 class Tree
126 {
127     int count;
128     int countSubtreesWithSumX(Node root, int X)
129     {
130         //Add your code here.
131         count = 0;
132         countSubTrees(root, 0, X);
133         return count;
134     }
135     int countSubTrees(Node root, int sum, int remain){
136         if(root==null)
137             return 0;
138         int data = root.data;
139         int left = countSubTrees(root.left, sum+data, remain-data);
140         int right = countSubTrees(root.right, sum+data, remain-data);
141         if(left+right+data==sum+remain)
142             count++;
143         return data+left+right;
144     }
145 }
146
```

18. Serialize and Deserialize a Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

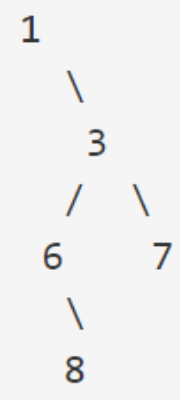
Clarification: The input/output format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

```
*/
public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    void buildString(TreeNode root, StringBuilder sb){
        if(root==null){
            sb.append("X"+" ");
            return;
        }
        sb.append(root.val+" ");
        buildString(root.left, sb);
        buildString(root.right, sb);
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Queue <String> q = new LinkedList(Arrays.asList(data.split(" ")));
        return buildTree(q);
    }

    TreeNode buildTree(Queue<String> q){
        String s = q.poll();
        if(s.equals("X"))
            return null;
        TreeNode root = new TreeNode(Integer.parseInt(s));
        root.left = buildTree(q);
        root.right = buildTree(q);
        return root;
    }
}
```

19. Node at distance

Given a Binary Tree and a positive integer **k**. The task is to count all distinct nodes that are distance k from a leaf node. A node is at k distance from a leaf if it is present k levels above the leaf and also, is a direct ancestor of this leaf node. If k is more than the height of Binary Tree, then nothing should be counted.

Input:



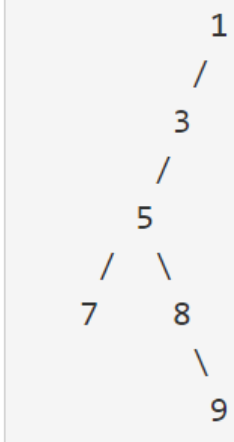
K = 2

Output: 2

Explanation: There are only two unique nodes that are at a distance of 2 units from the leaf node. (node 3 for leaf with value 8 and node 1 for leaves with values 4, 5 and 7)
Note that node 2 isn't considered for leaf with value 8 because it isn't a direct ancestor of node 8.

Example 2:

Input:



K = 4

Output: 1

Explanation: Only one node is there which is at a distance of 4 units from the leaf node.(node 1 for leaf with value 9)

Solution 1:

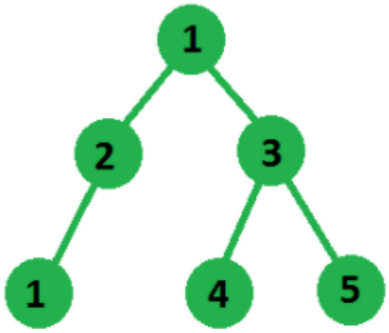
```
116 class Solution{
117     HashSet<Node> hs;
118     int printKDistantfromLeaf(Node root, int k)
119     {
120         // Write your code here
121         hs = new HashSet<Node>();
122         function(root, k);
123         return hs.size();
124     }
125     void function(Node root, int k){
126         if(root==null)
127             return;
128         if(f(root, k))
129             hs.add(root);
130         function(root.left, k);
131         function(root.right, k);
132     }
133     boolean f(Node root, int k){
134         if(root==null)
135             return false;
136         if(root.left==null && root.right==null && k==0)
137             return true;
138         return f(root.left, k-1) || f(root.right, k-1);
139     }
140 }
```

Solution 2: Space optimized

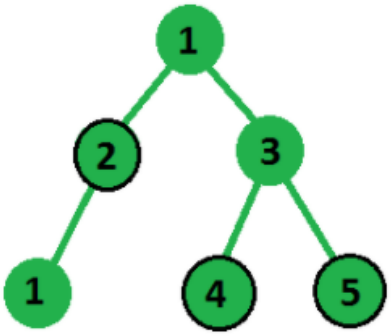
```
/* Given a binary tree and a nuber k, print all nodes that are k
distant from a leaf*/
int printKDistantfromLeaf(Node node, int k)
{
    if (node == null)
        return -1;
    int lk = printKDistantfromLeaf(node.left, k);
    int rk = printKDistantfromLeaf(node.right, k);
    boolean isLeaf = lk == -1 && lk == rk;
    if (lk == 0 || rk == 0 || (isLeaf && k == 0))
        System.out.print(" " + node.data);
    if (isLeaf && k > 0)
        return k - 1; // leaf node
    if (lk > 0 && lk < k)
        return lk - 1; // parent of left leaf
    if (rk > 0 && rk < k)
        return rk - 1; // parent of right leaf

    return -2;
}
```

20. Maximum sum of Non-adjacent nodes

Given a binary tree with a value associated with each node, we need to choose a subset of these nodes such that sum of chosen nodes is maximum under a constraint that no two chosen node in subset should be directly connected that is, if we have taken a node in our sum then we can't take its any children in consideration and vice versa.



Input Binary tree



Chosen nodes with maximum sum

```
Input:
        1
      /   \
     2     3
    /     /  \
   4     5    6
Output: 16
Explanation: The maximum sum is sum of
nodes 1 4 5 6 , i.e 16. These nodes are
non adjacent.
```

Solution:

We can solve this problem by considering the fact that both node and its children can't be in sum at the same time, so when we take a node into our sum we will call recursively for its grandchildren or when we don't take this node we will call for all its children nodes and finally we will choose maximum from both of these results.

It can be seen easily that the above approach can lead to solving the same subproblem many times, for example in the above diagram node 1 calls node 4 and 5 when its value is chosen and node 3 also calls them when its value is not chosen so these nodes are processed more than once. We can stop solving these nodes more than once by memoizing the result at all nodes.

In the below code, a map is used for memoizing the result which stores the result of the complete subtree rooted at a node in the map so that if it is called again, the value is not calculated again instead stored value from the map is returned directly.

```
119 class GFG
120 {
121     static int getMaxSum(Node root)
122     {
123         // add your code here
124         if(root==null)
125             return 0;
126         int inc = root.data;
127         if(root.left!=null)
128             inc += getMaxSum(root.left.left) + getMaxSum(root.left.right);
129         if(root.right!=null)
130             inc += getMaxSum(root.right.left) + getMaxSum(root.right.right);
131 
132         int exd = getMaxSum(root.left) + getMaxSum(root.right);
133 
134         return Math.max(inc, exd);
135     }
136 }
```

Same solution in another way:

```
public class FindSumOfNotAdjacentNodes {

    // method returns maximum sum possible from subtrees rooted
    // at grandChildrens of node 'node'
    public static int sumOfGrandChildren(Node node, HashMap<Node,Integer> mp)
    {
        int sum = 0;
        //  call for children of left child only if it is not NULL
        if (node.left!=null)
            sum += getMaxSumUtil(node.left.left, mp) +
                   getMaxSumUtil(node.left.right, mp);

        //  call for children of right child only if it is not NULL
        if (node.right!=null)
            sum += getMaxSumUtil(node.right.left, mp) +
                   getMaxSumUtil(node.right.right, mp);
        return sum;
    }
```

```
    //  Utility method to return maximum sum rooted at node 'node'
    public static int getMaxSumUtil(Node node, HashMap<Node,Integer> mp)
    {
        if (node == null)
            return 0;

        // If node is already processed then return calculated
        // value from map
        if(mp.containsKey(node))
            return mp.get(node);

        //  take current node value and call for all grand children
        int incl = node.data + sumOfGrandChildren(node, mp);

        //  don't take current node value and call for all children
        int excl = getMaxSumUtil(node.left, mp) +
                   getMaxSumUtil(node.right, mp);

        //  choose maximum from both above calls and store that in map
        mp.put(node,Math.max(incl, excl));

        return mp.get(node);
    }
```

```
    // Returns maximum sum from subset of nodes
    // of binary tree under given constraints
    public static int getMaxSum(Node node)
    {
        if (node == null)
            return 0;
        HashMap<Node,Integer> mp=new HashMap<>();
        return getMaxSumUtil(node, mp);
    }

    public static void main(String args[])
    {
        Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.right.left = new Node(4);
        root.right.right = new Node(5);
        root.left.left = new Node(1);
        System.out.print(getMaxSum(root));
    }
}
```

```
/* A binary tree node structure */
class Node
{
    int data;
    Node left, right;
    Node(int data)
    {
        this.data=data;
        left=right=null;
    }
};
//This code is contributed by Gaurav Tiwari
```

21. Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
inorder = [9,3,15,20,7]
postorder = [9,15,7,20,3]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

```java
class Solution {
    int postindex;
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        postindex = postorder.length-1;
        return fun(inorder, postorder, 0, inorder.length-1);
    }

    TreeNode fun(int []inorder, int []postorder, int is, int ie){
        if(is>ie)
            return null;

        TreeNode root = new TreeNode(postorder[postindex--]);
        int inindex = 0;
        for(int i=is;i<=ie;i++)
            if(inorder[i]==root.val){
                inindex = i;
                break;
            }
        root.right = fun(inorder, postorder, inindex+1, ie);
        root.left = fun(inorder, postorder, is, inindex-1);
        return root;
    }
}
```

22. Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

```java
class Solution {
    int preindex = 0;
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return fun(preorder, inorder, 0, inorder.length-1);
    }
    TreeNode fun(int[]preorder, int[]inorder, int is, int ie){
        if(is>ie)
            return null;

        TreeNode root = new TreeNode(preorder[preindex++]);

        int inindex=0;
        for(int i=is;i<=ie;i++){
            if(inorder[i]==root.val){
                inindex = i;
                break;
            }
        }
        root.left = fun(preorder, inorder, is, inindex-1);
        root.right = fun(preorder, inorder, inindex+1, ie);
        return root;
    }
}
```